Support exécutif scalable pour les architectures hybrides distribuées

Marc Sergent

Université Bordeaux 1

Mémoire de recherche

Master 2 Réseau, Systèmes et Mobilité
Spécialité Calcul Haute Performance

Support exécutif scalable pour les architectures hybrides distribuées

Auteur :
Marc Sergent

Encadrants :
Olivier Aumage
Samuel Thibault

Effectué à
Inria Bordeaux - Sud-Ouest

23 juin 2013

Table des matières

Chapitre 1

Introduction

1.1 Des machines de plus en plus complexes

Aujourd'hui, les accélérateurs sont pleinement entrés dans le monde du calcul haute performance (HPC). Les machines se complexifient et deviennent de plus en plus hétérogènes, car elles regroupent des types d'unités de calcul de plus en plus variées : accélérateurs graphiques (GPU), Cell (PS3), manycore, etc, en opposition aux machines dites homogènes ne contenant que des processeurs (CPU).

Des solutions pour exploiter pleinement ces machines hétérogènes ont été proposées, et des interactions ont été créées entre les différentes solutions d'utilisation de chaque ressource de calcul : multi-threading, OpenMP, OpenCL, CUDA, etc.

Des solutions pour faire communiquer des machines entre elles par le réseau ont également vues le jour, dans le but de pouvoir distribuer le travail sur plusieurs machines différentes : ce genre d'exécution est dite « distribuée ». La plus connue de toutes est le passage de messages, définie dans le standard MPI.

1.2 Hétérogénéité et distributivité : deux nécessités difficilement conciliables

Il est difficile pour le programmeur de concilier l'utilisation de toutes les solutions habituelles d'exploitation des machines hétérogènes, encore plus sur des grappes de machines potentiellement hétérogènes en elles-même, mais également entre elles.

La nécessité d'une couche logicielle portable permettant de décharger cette gestion complexe des épaules du programmeur est devenue indéniable : cette couche est appelée « support d'exécution ». En particulier, nous allons nous intéresser à StarPU, qui est le support d'exécution développé à Inria Bordeaux Sud-Ouest dans l'équipe RUNTIME.

De plus, les problèmes à traiter deviennent de plus en plus gros, et de plus en plus difficiles à gérer, même pour les supports d'exécution : il est donc nécessaire de trouver des solutions de passage à l'échelle.

1.3 Améliorer la scalabilité sur une grappe de machines hétérogènes

Le but du travail que nous allons présenter dans ce mémoire est d'améliorer la scalabilité de la surcouche de StarPU dédiée à la gestion de l'exécution sur des grappes de machines hétérogènes, nommée StarPU-MPI. De cet objectif découle plusieurs problématiques :

- La nécessité de se doter d'outils permettant d'évaluer les performances du support d'exécution, afin de pouvoir estimer les gains et/ou pertes introduits par notre travail.
- La nécessité des outils permettant d'analyser les performances obtenues pour pouvoir trouver de possibles causes de pertes de performance.
- L'isolation et l'apport de solutions aux problèmes de performance de StarPU-MPI mis à jour.
- La validation des solutions proposées par une nouvelle évaluation des performances.

Nous présentons dans le chapitre 2 d'autres supports d'exécution proposés par la communauté, leurs spécificités, et en quoi ils diffèrent de StarPU. Le chapitre 3 introduit le cadre de travail et d'expérimentation que nous avons élaboré, permettant la fiabilité et la reproductibilité des performances obtenues. L'étude des résultats expérimentaux, leur analyse et les propositions de solutions aux problèmes de performance constatés est présentée dans le chapitre 4. Enfin, nous présentons dans le chapitre 5 un bilan du travail effectué, ainsi qu'une discussion autour des perspectives d'amélioration de la scalabilité de StarPU sur des grappes de machines hétérogènes.

Chapitre 2

Maîtriser et combiner des paradigmes variés

De nombreux travaux sont menés pour essayer de résoudre le problème de la gestion de l'hétérogénéité grandissante des machines actuelles et à venir, passant du paradigme de threads simple à l'utilisation d'OpenMP, puis des langages spécifiques aux accélérateurs comme OpenCL et CUDA, tout en les combinant avec des paradigmes distribués comme MPI.

Cependant, l'utilisation simultanée de tous ces outils rend le travail de programmation très complexe, tant pour les questions d'équilibrage de charge que de gestion de la mémoire.
Pour décharger ce travail des épaules du programmeur, une couche logicielle supplémentaire, accessible sous forme d'une bibliothèque, a été introduite entre le système et l'application, chargée de gérer l'ordonnancement de cette dernière de manière transparente et performante : c'est ce qu'on appelle un support d'exécution. StarPU, sur lequel nous allons travailler, est l'un d'entre eux.

Dans un premier temps, nous allons présenter plusieurs paradigmes de supports d'exécution existants. Chacun adresse un sous-ensemble de problèmes spécifique : certains gèrent les machines hétérogènes uniquement, d'autres ont un support pour les exécutions distribuées mais ne gèrent pas complètement les machines hétérogènes. Les derniers, enfin, peuvent gérer les deux types d'exécutions, mais chacun a ses spécificités.

Dans un second temps, nous justifierons l'intérêt de StarPU par rapport à l'existant, puis nous présenterons son fonctionnement sur machine hétérogène, ainsi que sur des grappes de machines grâce à sa couche dédiée : StarPU-MPI.

2.1 De la gestion de machines hétérogènes ...

Le projet FLAME[1] propose la bibliothèque `libflame` ainsi que le support d'exécution SuperMatrix. Ces deux éléments permettent d'implémenter des applications d'algèbre linéaire et de les ordonnancer sur un seul noeud hétérogène pouvant disposer de plusieurs GPUs. Des algorithmes basiques d'ordonnancement sont proposés : simple file centrale, plusieurs files avec de l'association par affinité, plusieurs files avec du vol de travail, ainsi que des variantes avec priorités de ces algorithmes. Cependant, FLAME n'implémente pas de modèles de coût des transferts de données et de tâches.

Pour pouvoir faire de l'équilibrage de charge dynamique sur un seul noeud hétérogène CPU/GPU, l'approche que Qilin[2] a choisie est basée sur la construction du graphe de tâches en optimisant sa distribution via une étape de compilation dynamique supplémentaire au début de l'exécution. Cette étape d'optimisation utilise un historique des précédentes exécutions de l'application pour décider de la distribution du travail sur les différentes ressources de calcul. Cependant, cette opération se limite à l'optimisation d'un seul noyau de calcul par application.

L'approche choisie par le support d'exécution XKaapi[3] est d'implémenter l'équilibrage de charge et l'ordonnancement de tâches avec un système basé sur du vol de travail. Ce support d'exécution a été étendu pour supporter l'ordonnancement sur des machines hétérogènes multi-GPUs. Cette extension implémente une liste d'affinités pour favoriser le positionnement de données ayant de fortes interactions entre elles sur la même unité de calcul, ainsi qu'un mécanisme de seuil dynamique permettant de décider s'il est rentable d'exécuter une tâche sur un GPU ou non.

2.2 ... aux grappes de machines hétérogènes

GMH [4] introduit des communications explicites entre tous les GPUs de la grappe de machines sur laquelle il exécute l'application : les GPUs sont les nœuds MPI.

Les langages PGAS comme UPC ou XcalableMP[5, 6] permettent l'utilisation d'une mémoire partagée et distribuée. Certains disposent également d'une extension à la mémoire embarquée de certains GPUs. Les interfaces qu'ils proposent sont fortement guidées par l'application, et ne permettent pas d'ordonnancement dynamique.

Le support d'exécution DAGuE[7], récemment renommé PaRSEC, implémente un ordonnanceur distribué. Sa particularité est de proposer une méthode d'ordonnancement basée sur une représentation concise et algébrique du graphe de tâches, permettant ainsi de ne pas le dérouler entièrement en mémoire. Cette approche offre des avantages de scalabilité sur des grappes de machines, mais ne fonctionne pas pour les applications dont le graphe de tâches est calculé durant l'exécution.

Le support d'exécution Charm++[8] est basé sur un modèle de programmation orienté objets distribués, permettant le support de grappes de machines hétérogènes. Les méthodes d'entrée des objets sont étendues par des méthodes d'entrée accélérées, permettant d'indiquer le travail pouvant potentiellement être exécuté sur des accélérateurs. Au niveau ordonnancement, Charm++ implémente un gestionnaire d'accélérateurs pour coordonner l'exécution des méthodes d'entrée accélérées en utilisant de la surveillance de métriques de performance avec des méthodes d'équilibrage de charge.

Le projet ParalleX[9] présente un nouveau modèle de programmation parallèle et distribué, basé sur le concept de files de travail orientées message combinées avec un espace d'adressage actif global pour simplifier l'équilibrage de charge. Ce modèle est implémenté par le support d'exécution HPX, associé au projet.

La famille de langages StarSs[10] inclut une implémentation particulière appelée ClusterSs, permettant un support des grappes de machines hétérogènes très proche de celui de StarPU.

Cependant, les politiques d'ordonnancement proposées par ClusterSs sont encore basiques, et ne permettent pas d'exploiter toute la puissance des machines hétérogènes. De plus, ClusterSs impose l'utilisation d'un compilateur dédié pour la description des tâches, ce qui complexifie son intégration dans les applications existantes.

2.3 Discussion

Nous venons de présenter plusieurs paradigmes de supports d'exécution existants, chacun adressant un sous-ensemble de problèmes précis, et ayant leurs spécificités.

Cependant, chacun d'entre eux disposent également d'inconvénients ne permettant pas de proposer de la fonctionnalité nécessaire à notre travail. En effet, notre travail nécessite un support d'exécution qui peut exécuter des applications sur des grappes de machines hétérogènes, tout en offrant un ordonnancement dynamique performant. Il serait possible d'associer plusieurs d'entre eux pour obtenir un support d'exécution ayant ces fonctionnalités, comme XKaapi et un langage PGAS par exemple, mais cela complexifierait grandement les solutions à apporter.

C'est pourquoi nous avons choisi d'utiliser StarPU car, en plus d'être développé dans l'équipe Runtime, il offre ces fonctionnalités dans un seul support d'exécution, et se trouve donc être la cible idéale pour notre travail.

2.4 StarPU : ordonnancement de tâches sur machines hétérogènes

L'objectif principal d'un support d'exécution est de permettre d'exploiter pleinement les machines hétérogènes de manière totalement transparente pour le programmeur.

Le support d'exécution que nous allons étudier, StarPU, se base sur l'ordonnancement de graphes dynamiques de tâches de manière efficace sur machines hétérogènes[11, 12]. Son fonctionnement est basé sur un système de codelets, de dépendances de données implicites et/ou explicites et de soumission de tâches. Il dispose d'une mémoire virtuellement partagée permettant de réduire la quantité de transferts de données entre les différentes mémoires. Il propose également des modèles adaptatifs de prédiction de coût des calculs et des transferts de données.

StarPU est aussi une plateforme permettant l'expérimentation de stratégies d'ordonnancement. Les stratégies de StarPU auxquelles nous allons nous intéresser sont les suivantes :

- Ordonnanceur *eager* : simple ordonnanceur glouton
- Ordonnanceurs de la classe *dmda*. Un ordonnanceur dit de la classe *dmda* est un ordonnanceur qui cherche à minimiser le temps de terminaison de chaque tâche. Ce temps est estimé par la formule $(\alpha * Tcalcul) + (\beta * Ttransfert)$. Des variantes sont également à disposition : *dmdas* (supporte des priorités arbitraires) et *dmdar* (trie les tâches en fonction du nombre de données prêtes).

Il est également possible de créer son propre ordonnanceur, puis de l'intégrer et de l'utiliser au sein de StarPU.

2.5 StarPU-MPI : extension de StarPU au travail distribué

Il est maintenant nécessaire de se poser la question : comment StarPU fait-il pour gérer les exécutions sur des grappes de machines ?

Afin de pouvoir intégrer cette gestion, StarPU se base sur le paradigme de MPI. Il est ainsi possible d'étendre les applications StarPU sur des grappes de machines, mais également de pouvoir adapter aisément les applications MPI en applications StarPU distribuées. Il a été choisi de faire tourner une instance de StarPU par nœud, et d'utiliser une couche supplémentaire basée sur MPI pour faire communiquer les instances de StarPU entre elles. Cette couche a été nommée StarPU-MPI[13].

Pour pouvoir projeter le graphe de tâches sur une grappe de machines, nous avons mis en place une méthodologie basée sur une distribution statique des données décidée par l'application En effet, la complexité de la distribution des données d'une application sur une grappe de machines est telle que la définition statique de leur répartition est nécessaire pour les applications utilisant StarPU. La gestion des transferts de données explicites entre nœuds demandés par l'application est possible grâce à une API semblable à celle de MPI intégrée dans StarPU-MPI.

Lors de l'exécution d'un programme utilisant StarPU-MPI, toutes les instances de StarPU connaissent l'intégralité du graphe de tâches, ainsi que les propriétaires de chaque donnée. Grâce à cette connaissance, StarPU est capable, en fonction des dépendances de données implicites de chaque tâche, de substituer une dépendance implicite d'une tâche à une donnée qu'il sait ne pas détenir par la soumission d'une requête de réception correspondante. De même, comme chaque instance de StarPU connaît tout le graphe de tâches, il peut détecter qu'une tâche qui s'exécute sur un nœud distant va avoir besoin d'une donnée dont il est propriétaire, et initier l'envoi de la donnée vers ce nœud distant.

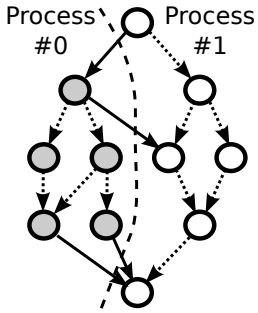


FIGURE 2.1 – Exemple de distribution du graphe de tâches entre 2 processus

Afin de pouvoir évaluer de manière fiable, reproductible, et pertinente les performances de la solution actuellement choisie dans StarPU, ainsi que de celle que nous allons développer, nous allons tout d'abord définir le cadre de travail et d'expérimentation dans lequel nous allons nous inscrire.

Chapitre 3

Cadre de travail et d'expérimentation

De manière générale, il est difficile de trouver les origines de problèmes constatés lors d'exécutions sur des grappes de machines hétérogènes. En effet, la complexité de programmation d'applications performantes sur ces machines rend le travail de débogage tout aussi difficile. Pour garantir la validité, la fiabilité et la reproductibilité des performances de StarPU, il est important de définir de manière précise et rigoureuse deux éléments :

- Le cadre de travail : sur quoi nous allons travailler, avec quels outils.
- Le cadre d'expérimentation : les résultats sont obtenus dans quelles conditions expérimentales, sur quelles applications, avec quels paramètres.

Nous allons commencer par présenter nos outils de travail et d'expérimentation : pour étudier les performances de StarPU, nous allons l'analyser grâce aux outils Pajé et Vite, et nous allons tester ses performances dans le cadre de la factorisation de Cholesky[14] présente dans la bibliothèque MORSE.

Nous allons ensuite dresser un premier bilan des performances de StarPU sur tous les types de machines et grappes de machines que nous avons à notre disposition sur PlaFRIM, afin de mettre à jour les possibles problèmes de performance.

3.1 Évaluer les performances : MORSE

La bibliothèque MORSE (Matrix Over Runtime Systems on Exascale[15]) est un recueil de noyaux d'algèbre linéaire. Ses points forts sont sa facilité d'utilisation et de mise en place ainsi que les très bonnes performances qu'elle expose. Elle peut être utilisée avec ou sans accélérateurs GPU, en s'appuyant sur MAGMA[16]. Il est également possible de l'associer à d'autres supports d'exécution, comme Quark[17].

Nous avons choisi d'utiliser cette bibliothèque afin de pouvoir bénéficier de noyaux et d'applications d'algèbre linéaire déjà écrites, testées et éprouvées, et ainsi obtenir des performances fiables et reproductibles.

Dans le cadre de notre travail, nous avons choisi de nous focaliser sur la factorisation de Cholesky

tuilée en flottants simple précision car elle possède toutes les caractéristiques typiques des algorithmes de résolution de problèmes d'algèbre linéaire dense.

Il est également nécessaire de pouvoir disposer d'outils nous permettant de comprendre le fonctionnement concret de StarPU pendant l'exécution d'une application. Des outils effectuant ce travail ont déjà été implantés dans StarPU, nous allons les réutiliser, puis les étendre selon nos besoins.

3.2 Des outils de visualisation d'exécution : Pajé et ViTE

Le format Pajé est un format de fichier de trace textuel et générique[18]. Il est dédié à la description du déroulement de programmes parallèles et/ou distribués. Son fonctionnement est basé sur la notion d'événement et de conteneur d'événement. Cependant, ce format ne donne que de l'information brute, dure à visualiser et à interpréter en l'état.

Une solution d'organisation et/ou de visualisation claire et intuitive de ces informations est donc nécessaire. L'outil de trace d'exécution que nous allons utiliser, qui permet de visualiser des informations à partir d'un fichier Pajé, est appelé ViTE (Visual Trace Explorer[19]). Son affichage est très intuitif, permettant une prise en main facile du logiciel. Il dispose en outre de fonctionnalités d'analyse plutôt intéressantes : zoom, coloration particulière de chaque type d'événement, affichage d'informations sur les transferts de données entre éléments d'un nœud, entre nœuds possible.

StarPU disposant déjà d'une implémentation complète des événements Pajé, nous allons nous baser sur celle-ci pour nos travaux, puis ajouter des événements qui peuvent nous être utiles au fur et à mesure.

Maintenant que nous avons défini le cadre de travail et d'expérimentation, nous pouvons présenter le protocole de test que nous avons mis en place pour analyser les performances de StarPU, afin de pouvoir mettre en lumière les points de contentions et/ou problèmes de scalabilité.

3.3 Un premier bilan : batterie de tests MORSE

3.3.1 Plateforme d'expérimentation : PlaFRIM

La plateforme PlaFRIM (Plateforme Fédérative pour la Recherche en Informatique et Mathématiques) est une plateforme dédiée à la recherche, et destinée à développer les capacités de modélisation, de simulation, de développement logiciel et d'expérimentation en mathématiques appliquées et en informatique. Nous allons présenter les machines de cette plateforme que nous avons utilisées pour obtenir les résultats présentés dans la suite de ce mémoire :

1. Les Fourmi : 68 machines homogènes 8 cœurs CPU. Nous utiliserons ces machines pour les expérimentations en homogène distribué, grâce au grand nombre de machines disponibles.
   - 2 Quad-core Nehalem Intel Xeon X5550 @ 2.66GHz
   - 24 Go de RAM
   - Sur réseau Infiniband

2. Les Mirage : 9 machines hétérogènes 12 cœurs CPU + 3 GPUs. Seules 8 des 9 machines sont identiques, la dernière ayant des GPUs différents. Nous allons donc utiliser cette grappe de 8 machines pour nos expérimentations en hétérogène et hétérogène distribué, mais également pour certaines de nos expériences en homogène, en désactivant les GPUs.
   – 2 Hexa-core Westmere Intel Xeon X5650 @ 2.67GHz
   – 36 Go de RAM
   – Sur réseau Infiniband
   – 3 NVIDIA Tesla M2070 GPU @ 1.15GHz
   – Mémoire dédiée : 6 Go GDDR5

3. Minotaure : une machine homogène 160 cœurs. Nous utiliserons cette machine pour tester la scalabilité de StarPU sur une exécution mono-noeud homogène avec un grand nombre d'unités de calcul.
   – 20 Intel Xeon CPU E7-8837 @ 2.67 GHz
   – 630 Go de RAM

Maintenant que nous avons présenté les machines avec lesquelles nous allons travailler, nous pouvons détailler la méthodologie que nous avons choisie pour évaluer les performances de StarPU sur la factorisation de Cholesky simple-précision de MORSE.

3.3.2 Protocole de test

Le protocole de test que nous avons mis en place est le suivant :

- Identifier les paramètres importants du problème
- Définir, pour chacun d'entre eux, l'intervalle de valeurs à tester ainsi que le pas d'évolution entre deux tests successifs
- Fixer chaque paramètre en fonction des performances crêtes observées
- Effectuer un test de scalabilité, en faisant varier la taille du problème, en mono-nœud, puis sur une grappe de machines (homogènes ou hétérogènes, mais toutes les machines d'une grappe sont identiques).
- Définir des métriques permettant d'analyser la scalabilité à partir des résultats expérimentaux : concepts de scalabilité faible et forte, d'efficacité, puis d'iso-efficacité.

Une arborescence de dossier générable automatiquement a également été mise en place dans ce but. Elle dispose de plusieurs éléments nécessaires à la batterie de tests :

- Des scripts de compilation automatique de StarPU + MORSE pour les 5 combinaisons suivantes : CPU-only, CPU-only + MPI, CPU-GPU, CPU-GPU + MPI, build Minotaure (160 cœurs). Possibilité de compiler avec –with-fxt pour obtenir des traces d'exécution.
- Des scripts de test pour toutes les combinaisons. Leur utilisation est totalement identique à celle des exécutables de tests MORSE originaux.
- Des scripts de jeux de tests, avec sauvegarde des résultats des tests dans des dossiers/fichiers prévus à cet effet.
- Des scripts d'édition de traces et/ou de graphes de performances associés aux fichiers de performance.
- Des fichiers de jobs différés pour PlaFRIM ont été créés.

Nous allons maintenant définir les métriques d'évaluation du passage à l'échelle des performances d'une application. Les plus connues d'entre elles sont la scalabilité faible et forte.

La scalabilité faible d'une application consiste à analyser, pour une taille de problème par unité de calcul fixée, les performances de celle-ci en fonction du nombre d'unités de calcul. La scalabilité forte d'une application consiste à analyser, pour un nombre d'unités de calcul fixé, les performances de celle-ci en fonction de la taille du problème sur lequel elle travaille. Cette métrique est couramment utilisée dans les articles de recherche pour justifier la scalabilité des performances. Elle est nécessaire, mais pas suffisante. C'est pourquoi deux autres métriques ont été inventées, basées sur le principe d'efficacité des performances obtenues.

L'efficacité décrit le pourcentage de performances obtenues par rapport aux performances théoriques prises sur une unité de calcul. Cette métrique est souvent calculée comme suit : supposons une machine Mirage, ayant 12 CPUs et 3 GPUs, donc 3 CPUs dédiés au contrôle des GPUs. L'efficacité des performances sur Mirage sera donc calculée ainsi :
$Eff = Tcalcul_{obtenu}/((12-3) * Tcalcul_{monoCPU} + 3 * Tcalcul_{monoGPU})$.

La dernière métrique qui nous intéresse est l'iso-efficacité. Cette dernière permet de décrire la rapidité de la croissance des performances jusqu'à un taux d'efficacité fixé par l'expérimentateur. On la calcule en isolant, pour chaque nombre d'unités de calcul, la taille de matrice correspondant à la première valeur pour laquelle l'efficacité choisie est atteinte, puis en interpolant de manière polynomiale ces points afin d'en extraire une courbe. La nature de cette courbe permet d'obtenir une évaluation de l'iso-efficacité des performances en fonction du problème. Nous y reviendrons avec un exemple concret dans la section 3.3.4.

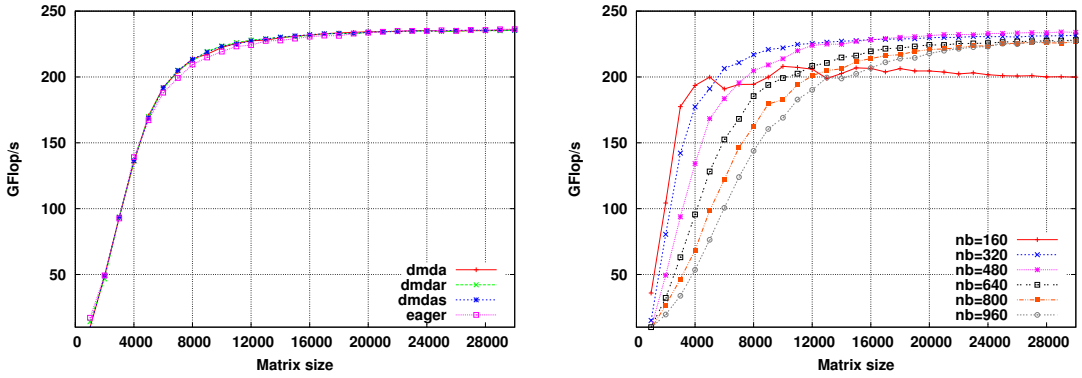
Maintenant que nous avons détaillé le protocole de test et les outils que nous avons mis en place pour automatiser, faciliter, et assurer la réutilisabilité du protocole et la reproductibilité des tests, nous pouvons présenter les premiers résultats que nous avons obtenus.

3.3.3 Premiers résultats

Les paramètres importants à déterminer lors de l'exécution d'une application ayant StarPU comme support d'exécution sont les suivants :

- Déterminer quel ordonnanceur est le plus efficace, entre l'ordonnanceur *eager* et les ordonnanceurs de la classe *dmda*.
- Déterminer quelle est la taille de tuilage la plus adaptée en fonction de l'application.
- Pour les exécutions sur machines disposant de GPUs, déterminer la valeur du paramètre β, servant à pondérer le coût des transferts mémoire entre le GPU et la mémoire centrale.

Les figures ci-après présentent les résultats obtenus sur des machines homogènes (Mirage en désactivant les GPUs, et Fourmi).

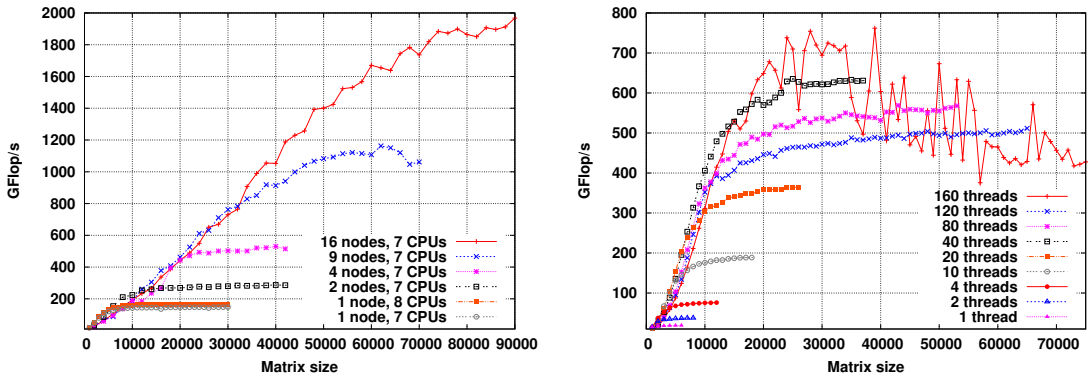


(a) Choix de l'ordonnanceur (Mirage) (b) Choix de la taille de tuile (Mirage)

FIGURE 3.1 – Choix des paramètres sur machine homogène

La figure 3.1(a) nous informe que les ordonnanceurs affichent tous des performances à peu près équivalentes sauf *eager*, dont l'efficacité maximale est atteinte plus tardivement que les autres. Nous discuterons dans la suite de ce mémoire de l'utilité de cet ordonnanceur, notamment pour les exécutions sur des machines homogènes disposant d'un grand nombre de cœurs.

La figure 3.1(b) montre l'impact que la taille de tuilage peut avoir sur les performances d'une exécution. Trop petites, les tuiles ne permettent pas aux cœurs d'exploiter tout leur potentiel de calcul, ce qui conduit à des pertes de performance. Trop grosses, l'application n'expose plus assez de parallélisme, et l'efficacité maximale n'est atteinte que pour des tailles de problème dépassant 20000*20000. La taille de tuilage qui nous paraît la plus intéressante est 480 car, tout en atteignant la performance crête en régime permanent, elle offre une efficacité acceptable pour les petits problèmes.



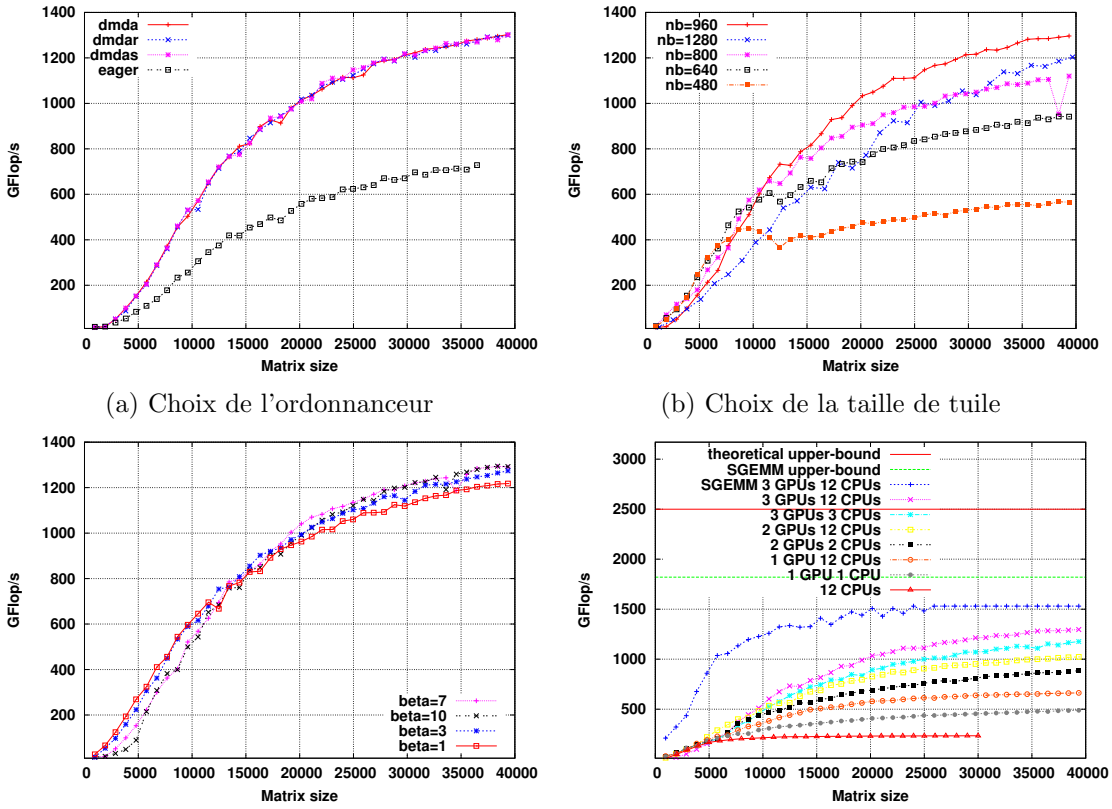
(a) Performances en distribué (Fourmi) (b) Scalabilité sur Minotaure

FIGURE 3.2 – Premiers résultats obtenus sur machine homogène

Sur la figure 3.2(a), nous présentons les performances obtenues sur des grappes allant jusqu'à 16 machines homogènes (Fourmi). Il est important de noter qu'un cœur est volontairement exclu, de sorte que le thread de communication MPI de StarPU puisse s'y attacher sans risque d'être dérangé, et ainsi assurer une progression correcte des communications. Les performances obtenues sont conformes aux attentes au niveau de la scalabilité, avec une efficacité de 85% sur 9 nœuds, et d'environ 80% sur 16 nœuds, ce qui est très convenable.

La figure 3.2(b) présente les premières performances obtenues sur la machine Minotaure (160 cœurs). Les résultats obtenus nous semblent problématiques : seule la courbe pour 160 cœurs dépassent celle obtenue avec 40 cœurs, et de manière très instable. Nous allons donc proposer des solutions dans la section suivante afin d'apporter une solution simple pour résoudre ce problème.

Les figures suivantes montrent les performances obtenues sur une machine hétérogène (Mirage).



(a) Choix de l'ordonnanceur

(b) Choix de la taille de tuile

(c) Choix du paramètre Bêta

(d) Performances en mono-nœud

FIGURE 3.3 – Premiers résultats obtenus en mono-nœud sur machine hétérogène

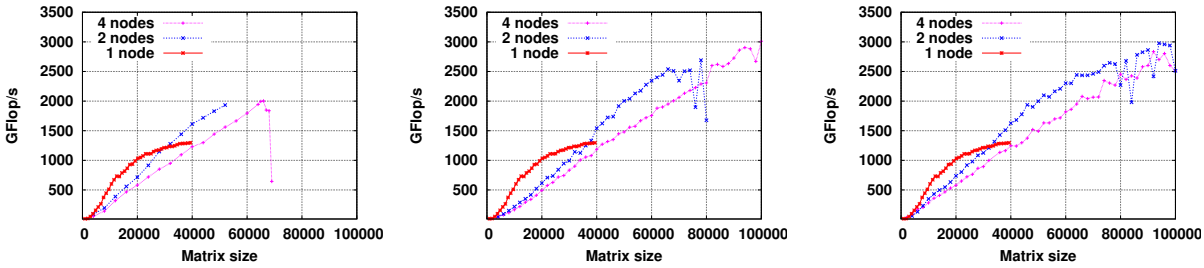
Les figures 3.3(a),(b) et (c) présentent, comme sur machine homogène, les tests effectués sur chaque paramètre, permettant de déduire les valeurs pour lesquelles nous obtenons les meilleures performances.

Pour la figure 3.3(a), il apparaît qu'un ordonnanceur de la classe *dmda* devient indispensable pour les performances dans le cadre d'une exécution sur machine hétérogène. La figure 3.3(b) nous apprend que, pour les tailles de problème dépassant 12000*12000, 960 est la taille de tuilage la plus performante. En effet, les GPUs ont besoin d'une taille de bloc suffisamment grande pour pouvoir exploiter toutes leurs capacités de calcul, mais aussi pour pouvoir recouvrir les temps de communication (notamment de copie de données) par du calcul. Il ne faut pas trop grossir les tuiles non plus, car l'application ne va plus exposer le parallélisme nécessaire à l'exploitation de toutes les ressources de calcul de la machine hétérogène. Pour le choix de Bêta sur la figure 3.3(c), le résultat présente moins de sensibilité. Il nous est cependant possible

d'affirmer qu'un Bêta en-dessous de 3 affecte sensiblement les performances sur les exécutions des petits problèmes, et qu'un Bêta au-dessus de 10 diminue légèrement les performances sur les gros problèmes. Nous avons donc choisi, dans nos tests futurs, de fixer Bêta à 7.

La dernière figure 3.3(d) présente les résultats que nous avons obtenus en mono-nœud (Mirage) sur la factorisation de Cholesky tuilée simple-précision, avec les trois paramètres fixés comme précisés ci-dessus, mais en faisant varier cette fois-ci le nombre de ressources utilisables. Pour référence, nous avons ajouté à la figure les courbes de performance obtenues avec le noyau SGEMM sur ces mêmes machines, ainsi que la borne supérieure du SGEMM et la borne théorique de la machine. Nous pouvons observer sur cette figure que la performance crête obtenue sur la factorisation de Cholesky est de 1300 GFlops, ce qui est très convaincant.

Les trois figures ci-dessous présentent les performances obtenues sur une grappe de machines hétérogènes.



(a) Taille de tuile = 1000 (b) Taille de tuile = 2000 (c) Nombre de tuiles = 1600

FIGURE 3.4 – Premiers résultats obtenus sur une grappe de machines hétérogènes

Nous constatons que les performances de scalabilité obtenues ne sont pas très bonnes, tant en terme d'efficacité que d'iso-efficacité. En particulier, nous remarquons que les performances sont très instables, parfois même l'exécution ne termine pas, à partir d'une certaine taille de problème. En analysant plus finement, nous avons compris que le problème n'est pas lié à la taille de la matrice, mais à la granularité de découpage de celle-ci. Nous avons pu estimer empiriquement le seuil que le nombre de tuiles doit dépasser pour faire apparaître le problème. Ce seuil se situerait autour de 80 tuiles par dimension, soit 6400 tuiles.

3.3.4 Des ajustements : numactl et *eager*

Dans la section précédente, nous avons pu effectuer un bilan complet des performances, qui ont permis de détecter des performances anormales sur des grappes de machines hétérogènes. Ces problèmes seront traités dans le chapitre 4.

Dans cette section, nous allons revenir sur le test de scalabilité effectué sur la machine Minotaure (figure 3.2(b)). Nous allons essayer de comprendre la source de ce manque de performance, proposer une solution simple, la déployer, puis reproduire le test afin d' observer les changements provoqués, ainsi que les performances obtenues. Pour comprendre ces problèmes de performance, il est important de présenter le comportement connu de StarPU sur machine homogène avec plusieurs nœuds NUMA, ainsi que de détailler plus précisément un point important du choix d'ordonnancement de l'ordonnanceur *dmda*.

StarPU possède un système de mémoire virtuellement partagée lui permettant de réduire le nombre de transferts de données entre les différentes mémoires. Actuellement, ce système ne permet pas la gestion efficace de la localité des données, notamment entre plusieurs nœuds NUMA. Il arrive donc que, lors d'un calcul de matrice par exemple, tout le problème soit alloué sur un seul nœud NUMA alors que la machine en possède plusieurs, ce qui va provoquer, même avec un bon ordonnanceur, de la contention mémoire sur le nœud NUMA possédant toutes les données.

En particulier, ce problème se pose sur la machine Minotaure, étant donné que son architecture est telle que les 160 cœurs de la machine sont répartis entre 8 nœuds NUMA. Pour apporter une solution au problème sans entamer des modifications profondes de StarPU qui seront l'objet de travaux futurs, nous avons choisi d'utiliser l'outil *numactl*, permettant de contrôler l'allocation de données des processus sur une machine possédant plusieurs nœuds NUMA. Dans le cadre de notre expérience, nous avons donc choisi d'ajouter `numactl --interleave=all` à notre commande d'exécution, afin d'assurer la distribution équitable des données sur tous les nœuds NUMA de la machine, assurant ainsi la répartition des requêtes mémoire sur les différents bancs mémoire.

Le deuxième problème concerne la méthode de choix de l'ordonnanceur *dmda*. En effet, cet ordonnanceur, pour faire son choix, cherche à « élire » une ressource sur laquelle effectuer une tâche considérée à un instant T. De ce fait, il doit donc parcourir l'intégralité des caractéristiques, à l'instant T donné, des ressources pour pouvoir en élire une. Sur une machine telle que Minotaure, cette stratégie montre ses désavantages car il est très coûteux de parcourir les informations de 160 ressources à chaque choix d'ordonnancement de tâche.

Pour pallier ce problème, nous avons choisi d'utiliser l'ordonnanceur *eager* car, contrairement à *dmda*, le temps pris par ses choix d'ordonnancement est indépendant du nombre de ressources de la machine considérée. De plus, d'après la figure 3.1(a), nous avons pu constater que leurs performances sont identiques en régime permanent, dès que la taille du problème est suffisamment grande.

Nous avons appliqué ces deux solutions, puis reproduit le test de scalabilité, et obtenu les résultats suivants :

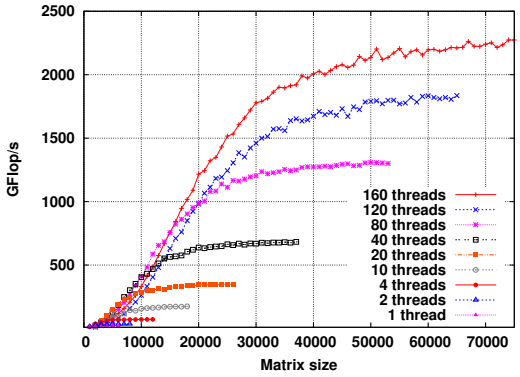
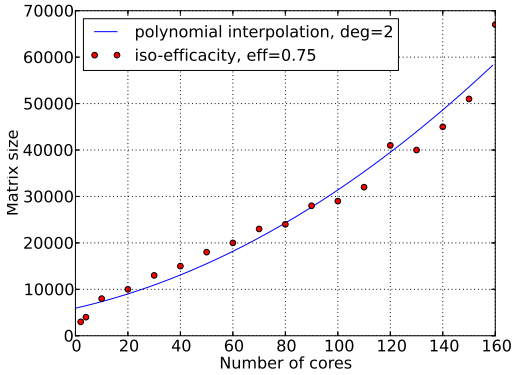
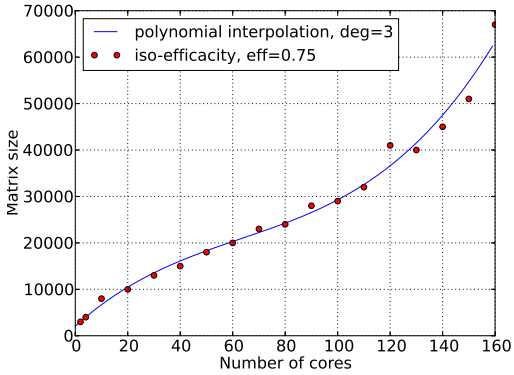


FIGURE 3.5 – Scalabilité sur Minotaure



(a) Courbe d'iso-efficacité à 75% de degré 2 de Minotaure



(b) Courbe d'iso-efficacité à 75% de degré 3 de Minotaure

FIGURE 3.6 – Résultats sur Minotaure avec numactl et *eager*

Sur les figures ci-dessus, nous pouvons constater que les performances obtenues sont désormais en accord avec les résultats auxquels nous nous attendions, car nous pouvons constater une évolution des performances en fonction du nombre d'unités de calcul, prouvant la scalabilité des performances. Pour étudier plus en détails les caractéristiques de scalabilité, nous avons dressé une courbe d'iso-efficacité à 75%, interpolée par des polynômes de degré 2 et 3.
La factorisation de Cholesky étant un algorithme d'algèbre linéaire dense effectuant, si N est la taille d'un côté de la matrice considérée, N^3 calculs sur N^2 données, la scalabilité idéale souhaitée est une scalabilité en N^3, soit la quantité de calcul. Or, nous pouvons constater que l'iso-efficacité à 75% est interpolable par un polynôme de degré 2, mais pas par un polynôme de degré 3, car on constate une inflexion de la courbe. Nous pouvons donc affirmer que StarPU permet, sur Minotaure, une scalabilité de la factorisation de Cholesky simple-précision à 75% en N^2, soit la taille des données, ce qui est très convenable.

3.3.5 Résultats de l'analyse

Dans la partie 3.3.3, nous avons présenté les premiers résultats que nous avons obtenus qui vont servir de base de référence pour notre travail, afin de pouvoir avoir un point de repère sur les performances de départ de StarPU.

Nous avons vu également, dans la partie 3.3.4, que certaines performances n'étaient pas satisfaisantes, et réussi à trouver des solutions pour certaines. Cependant, le problème de performances instables dans le cadre de l'utilisation de StarPU sur des grappes de machines hétérogènes reste entier.

Ces résultats, et ce problème subsistant, constituent l'origine ainsi que la justification du travail que nous allons présenter dans ce mémoire, afin d'améliorer la scalabilité de StarPU sur des grappes de machines hétérogènes.

Chapitre 4

Identification et résolution des problèmes de scalabilité

Dans ce chapitre, nous allons étudier, dans un premier temps, les traces d'exécution obtenues pour les cas problématiques, leur ajouter de l'information si besoin, pour ainsi mettre à jour les causes de ce manque de performances.

Dans un second temps, nous proposerons des solutions pour résoudre ces problèmes, et nous validerons la solution retenue par de nouveaux tests de performances.

4.1 Gestion des tags MPI dans StarPU

Le premier problème que nous allons traiter concerne l'utilisation des tags MPI dans StarPU, et plus particulièrement le nombre de tags autorisés dans les implémentations MPI actuelles, notamment dans OpenMPI.

4.1.1 Une utilisation intensive de MPI

Dans le standard MPI, un tag est une étiquette servant à identifier de manière unique une communication MPI. Dans les implémentations MPI, cette étiquette est souvent représentée sous la forme d'un entier signé. Par définition du standard, chaque implémentation MPI doit fournir le nombre de tags maximal supporté. L'accès a cette valeur se fait par une interface définie dans le standard : accès à l'attribut MPI_TAG_UB (section 8.1.2 du MPI-2.2[20]). Pour OpenMPI, cette valeur peut prendre n'importe quelle valeur positive sur un entier signé, donc $2^{31} - 1$ possibilités.

En revanche, le nombre de requêtes MPI actives au même instant est plus fortement limité dans toute bibliothèque MPI. Il y a une multitude de raisons à cela :
- Optimisation de l'usage mémoire avec les réseaux haut débit (type Infiniband)
- Compatibilité C-Fortran : chaque objet MPI (type de donnée, communicateur, fenêtre) doit avoir un identifiant unique car ils sont représentés par des entiers en Fortran.
- Optimisation de l'espace de recherche pendant l'opération de correspondance.

Dans OpenMPI, ce nombre est limité par défaut a 32768.
Afin de permettre le passage à l'échelle de StarPU, il est nécessaire de pouvoir dépasser cette

limitation, en intégrant la gestion des tags directement dans StarPU-MPI, pour assurer la fiabilité de la scalabilité sur les gros problèmes.

Dans l'état actuel des outils de visualisation d'exécution présents dans StarPU, il n'est pas possible de pouvoir observer le comportement du thread de communication MPI associé à chaque instance de StarPU-MPI durant l'exécution. Nous allons donc commencer par proposer des extensions pour ces outils afin de le permettre.

4.1.2 Amélioration des outils de visualisation

Pour permettre la visualisation des événements associés au comportement du thread de communication MPI, nous allons nous baser sur l'implémentation existante des conteneurs et événements Pajé dans StarPU, en proposant des extensions de celle-ci.

Nous avons commencé par créer un nouveau conteneur d'événements Pajé destiné aux événements associés au thread de communication MPI, que nous avons nommé MPI Communication Thread (MPICt). Nous avons ensuite décrit les différents événements associés aux états possibles du thread de communication MPI : donnée en cours d'envoi / envoyée, en cours de réception / reçue, etc. Pour finir, nous avons ajouté la chaîne d'appels de fonctions correspondant aux nouveaux événements, puis positionné les événements de trace dans le code de StarPU-MPI.

Voici un exemple de trace d'exécution obtenue avec ViTE après l'extension que nous avons proposée :

FIGURE 4.1 – Exemple de trace avec les informations de thread MPI

Maintenant que nous disposons d'un moyen d'observer le fonctionnement concret des communications MPI dans StarPU, nous pouvons commencer à préciser les modifications que nous avons apporté à StarPU-MPI pour gérer la grande quantité de tags à laquelle il peut faire appel.

4.1.3 Gestion des tags dans StarPU-MPI

Dans cette section, le problème qui nous concerne est le suivant : comment déporter la gestion de l'opération interne à MPI de correspondance des communications dans StarPU-MPI ?

La solution que nous proposons est basée sur un système d'enveloppe. Nous avons décidé d'étendre le système d'enveloppe présent dans StarPU-MPI, servant actuellement à envoyer la taille des données à recevoir dans le cadre d'un transfert de données dont le type est décrit par l'application. Dans la solution que nous proposons, l'enveloppe contient désormais le tag associé à la donnée qui va directement la suivre, car toutes les communications MPI se font sur un tag unique que nous avons nommé _starpu_mpi_tag(modifiable via des accesseurs). L'émetteur envoie donc systématiquement une enveloppe décrivant la donnée qui va arriver au récepteur. Côté récepteur, dès lors qu'au moins une requête de réception est en attente, une requête d'en-tête est postée en permanence, destinée à attraper les enveloppes qui arrivent. Dès qu'une enveloppe arrive, la requête de réception correspondante est récupérée depuis une table de hachage, pour être postée directement.

Pour implémenter cette table de hachage, afin de permettre une meilleure maintenabilité, nous avons retenu le système déjà utilisé dans StarPU : *UThash*. Lorsqu'une réception est postée, au lieu de simplement l'ajouter à la liste des requêtes en attente d'être postées par le thread de communication MPI, elle est maintenant ajoutée à une table de hachage dont la clé est le tag de la requête (= tag de la donnée), et la valeur la requête elle-même. Ainsi, lorsqu'une enveloppe arrive, le thread de communication MPI ira chercher dans la table de hachage la requête correspondant au tag contenu dans l'enveloppe pour la poster ensuite. Il est donc nécessaire que chaque donnée ait un tag unique : c'est à l'application de le garantir.

Un nouveau problème est introduit par notre solution : toutes les communications doivent désormais être « ordonnées » pour pouvoir convenir au récepteur, qui gère chaque une enveloppe correspondant à une donnée, puis de faire « attendre » la réception de la donnée (en attendant l'allocation de la donnée côté récepteur, par exemple), en faisant passer d'autres communications devant, car la prochaine réception d'enveloppe postée serait corrompue par la réception de la donnée correspondant à l'enveloppe « mise en attente », à cause du tag unique utilisé pour toutes les communications. Il faut donc pouvoir gérer le cas : réception d'une enveloppe annonçant l'arrivée de données pour lesquelles la réception n'a pas encore été postée par l'application.

L'idée que nous avons retenue pour dépasser cette difficulté est d'ajouter un système de stockage des données arrivant avant le postage de la requête correspondante dans des tampons temporaires, qui seront enregistrés dans une seconde table de hachage tag - donnée qui leur est dédiée. Lorsque la réception correspondante est alors postée par l'application, le thread de communication MPI sera en mesure de détécter si la donnée correspondante est déjà arrivée ou non, et si oui, fera une copie du tampon temporaire dans la donnée associée à la requête, au lieu de soumettre une requête.

Pour vérifier la robustesse de l'implémentation que nous avons proposé, nous avons effectué une campagne de tests basés sur ceux associés à StarPU-MPI dans le dépôt de StarPU, avec succès.

Nous allons expliquer plus précisément le fonctionnement concret du système enveloppe + donnée, en présentant de manière simplifiée l'algorithme correspondant dans le code implémenté.

4.1.4 Fonctionnement du système enveloppe + donnée

```
struct _starpu_mpi_envelope
{
    int mpi_tag;
    size_t taille_donnée;
}

[...]

static void* _starpu_mpi_progress_thread_func()
{
    [...]

    Tant que ( l'application s'exécute )
    {
        [...]

        - Gérer les requêtes déjà soumises
        - Soumettre la requête captive d'enveloppes
        - Gérer les requêtes détachées

        Si ( une enveloppe a été reçue )
        alors {
            - Chercher la requête correspondante dans la table de hachage
            - Soumettre la requête
        }
        sinon {
            [...]
        }
    }
}
```

Nous allons maintenant expliquer le fonctionnement particulier que nous avons mis en place dans le cas où une enveloppe reçue contient le tag d'une requête qui n'a pas encore été postée par l'application, tout en sachant que la donnée correspondant à l'enveloppe va arriver directement après celle-ci.

4.1.5 Une donnée entrante, sans réception encore postée

Pour gérer ce cas particulier, nous avons utilisé au maximum le système de dépendance implicite de données de StarPU. Dans notre solution, le thread de communication MPI poste sur le tampon temporaire dédié à la réception anticipée des données une requête `starpu_mpi_irecv_detached`, afin d'acquérir les droits en écriture sur ce dernier grâce au `starpu_data_acquire_cb` du `starpu_mpi_isend_irecv_common`, ce qui permet la poursuite de l'exécution.

De plus, dans la fonction `starpu_mpi_submit_new_request`, s'il est détecté que la requête qui est en cours de traitement est une requête postée par le thread de communication MPI, l'allocation du type de donnée MPI correspondant à la donnée, faite traditionnellement par le thread de communication MPI, est faite directement par le thread soumettant la requête, afin de pouvoir ajouter directement la requête à la liste des requêtes prêtes au lieu de l'ajouter dans la table de hachage.

L'intérêt étant que, dans le thread de communication MPI, les requêtes en attente dans la liste des requêtes prêtes sont traitées avant de se poser la question du postage (ou pas) de la requête d'en-tête destinée à attraper les enveloppes, ce qui permet au `MPI_Irecv` en bout de chaîne d'être posté avant celui correspondant à la requête d'en-tête, et préserve ainsi l'ordre des communications.

Lorsque la réception correspondante est alors soumise par l'application, et qu'elle détecte que la réception a déjà été postée en interne, elle va à la place effectuer une demande d'acquisition des données en lecture (`starpu_data_acquire_cb` en STARPU_R), dont la fonction de rappel (appelée lorsque la donnée est acquise) est une fonction effectuant la recopie des données du tampon temporaire vers la donnée associée à la requête.

4.1.6 Un gain en fiabilité

Dans cette section, nous avons présenté comment nous avons déporté la gestion des tags dans StarPU-MPI, en utilisant un seul et unique tag pour communiquer via la couche MPI (_starpu_mpi_tag). Après analyse des performances, nous n'avons pas constaté de gain, ni de perte de performances avec notre solution par rapport à la version originelle de StarPU-MPI. Cependant, nous nous sommes assurés que ce potentiel problème ne se produira jamais à l'avenir, et avons donc obtenu un gain de fiabilité substantiel de StarPU-MPI.

Nous avons présenté, dans cette partie, le premier problème auquel nous avons été confrontés, la solution que nous avons choisi, et les gains que nous avons obtenus. Cependant, nous devons maintenant traiter le second problème que nous avons mis à jour dans la partie 3.3.3, à savoir l'instabilité des performances de StarPU sur des grappes de machines hétérogènes.
Pour cela, nous allons analyser plus en détails le problème en étendant davantage les outils de visualisation d'exécution afin de mettre à jour la source du problème, proposer plusieurs solutions, puis présenter les performances que nous avons obtenues avec la solution que nous avons retenue.

4.2 Des performances instables sur des grappes de machines hétérogènes

Pour analyser plus en détails ce problème, nous allons utiliser les outils de visualisation d'exécution Pajé et ViTE, utilisables avec StarPU, afin de comprendre d'où vient le problème qui nous préoccupe.

4.2.1 Des transferts de données anormalement longs

Pour introduire le problème étudié, nous allons présenter une trace d'exécution mettant en évidence la nature du problème : des transferts de données vers le GPU au moment d'une communication MPI anormalement longs.

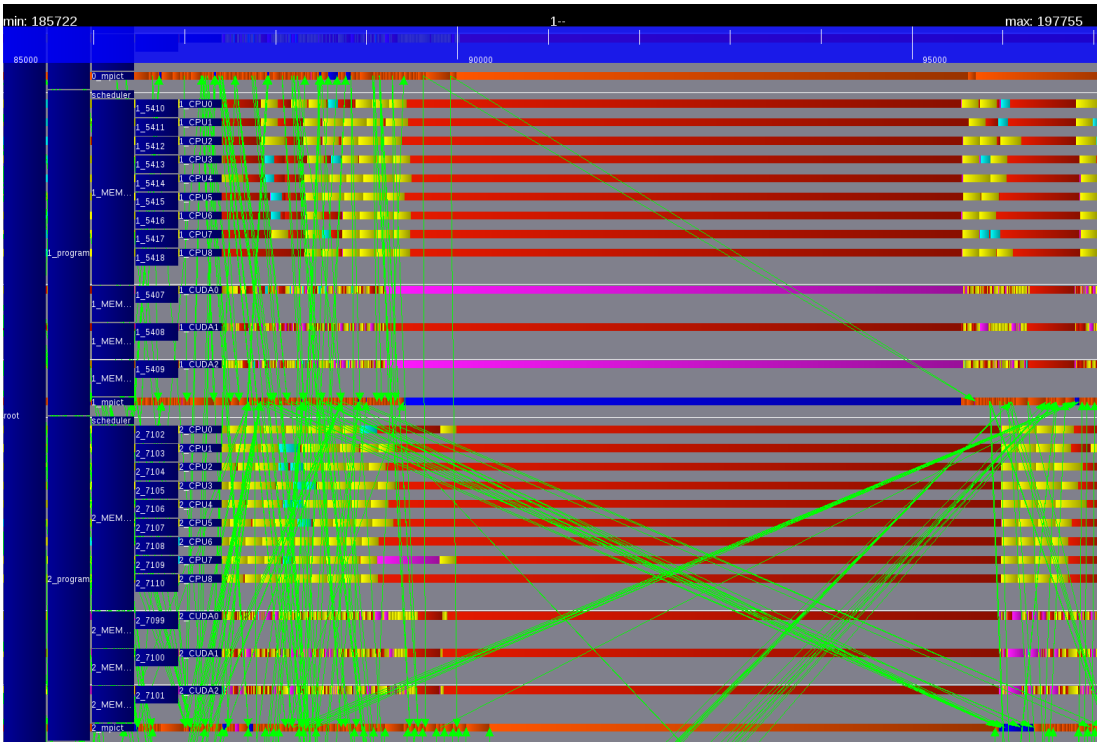


FIGURE 4.2 – N=100000,NB=1250,3TFlops

Les informations actuelles apportées par les événements de trace étant insuffisantes pour pouvoir déterminer la nature du problème, il est nécessaire d'ajouter de nouveaux événements de trace pour avoir une meilleure compréhension du problème.

4.2.2 Extension des outils de visualisation

Comme la nature du problème constaté est aléatoire, i.e que la fréquence des blocages, ainsi que leur durée sont totalement non déterministes, nous émettons une première hypothèse : ce problème peut être lié au comportement des mutex et/ou spinlocks. Nous ajoutons donc des événements correspondant aux prises/relâchements de ces derniers.

Les traces obtenues avec ces événements étant trop volumineuses, il nous est impossible de les analyser avec ViTE. À la place, nous allons faire du filtrage pour analyser directement les événements de trace, grâce à l'outil fxt_print de la bibliothèque FxT.

Grâce à cette analyse, nous avons pu déterminer la zone de code dans laquelle l'exécution reste confinée durant le problème : le gestionnaire de mémoire intégré de StarPU. Cela nous indique donc un possible problème de libération de la mémoire.

Pour confirmer cette nouvelle hypothèse, nous ajoutons des événements de trace avertissant d'un appel échoué à la fonction d'allocation de mémoire interne à StarPU, indiquant donc qu'une mémoire est pleine, de manière à pouvoir constater si cet événement est bien appelé de manière répétée pendant le problème. De plus, nous allons enlever les événements de trace concernant les verrous, de sorte à alléger le nombre d'événements dans la trace, et permettre de nouveau l'utilisation de ViTE pour visualiser le problème.

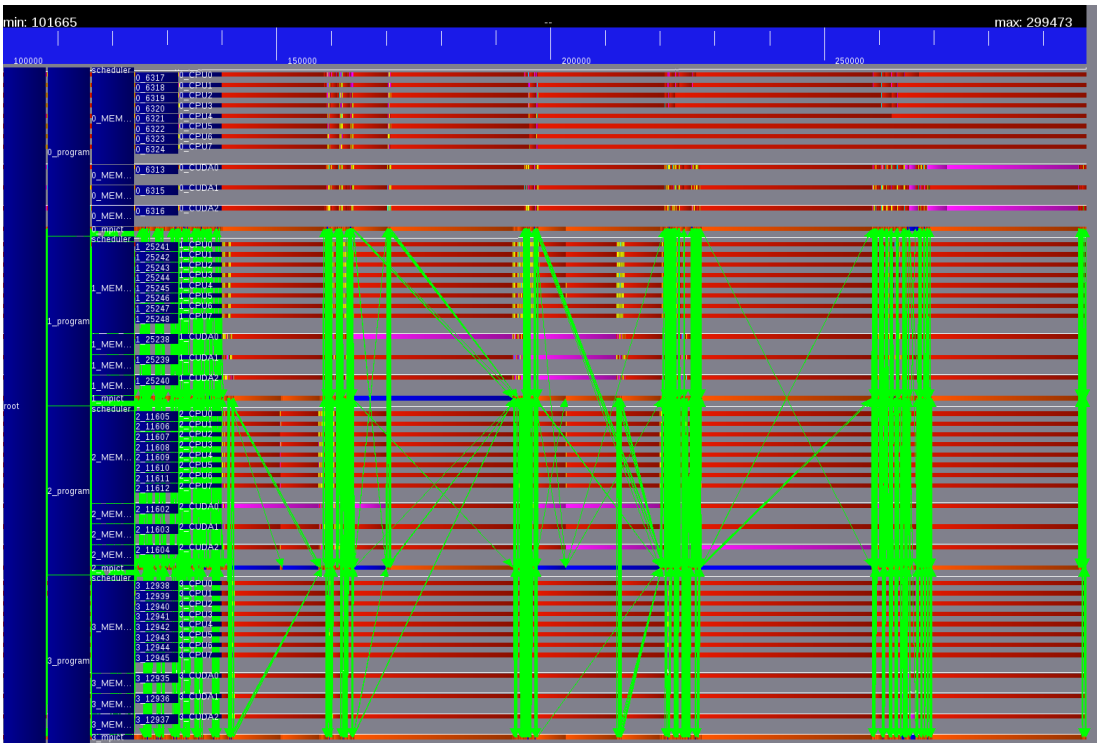


FIGURE 4.3 – Trace avec événements d'allocation mémoire échouée

Cette trace permet de mettre en exergue toutes les zones où le problème survient en les allongeant, ce qui le rend très facile à constater. De plus, cette trace met en évidence que le problème ne survient pas que lorsqu'une communication MPI et un transfert de données GPU coïncident, mais également lorsque plusieurs GPUs effectuent des transferts de données simultanés.

Cependant, nous ne savons toujours pas pourquoi l'exécution boucle dans cette zone. Nous allons de nouveau ajouter de l'information, en affichant dans les événements de trace les données concernées par ces échecs d'allocation, ainsi que leur taille.

En analysant ces informations supplémentaires, nous avons découvert l'origine du problème, qui se produit lors du scénario suivant : pour l'exemple, nous nommerons les deux workers GPUs qui y prennent part GPU1 et GPU2, et l'élément perturbateur sera le thread de communication MPI, qui sera nommé MPICt :

- MPICt prend le verrou sur la donnée X.
- GPU1 essaye d'exécuter une tâche sur la donnée X. Or, le spinlock header_lock sur la donnée X est verrouillé : GPU1 part précharger la donnée Y sur GPU2.
- MPICt relâche le verrou sur la donnée X.
- GPU2 essaye d'exécuter une tâche sur la donnée Y. Or, le spinlock header_lock sur la donnée Y est verrouillé par GPU1 : GPU2 part précharger la donnée X sur GPU1.
- Le chargement de Y par GPU1 échoue : refus d'allocation mémoire. GPU1 retourne essayer de prendre le verrou sur la donnée X. Or, le spinlock header_lock sur la donnée X est verrouillé par GPU2 : GPU1 part précharger de nouveau la donnée Y sur GPU2 (puisque le premier chargement a échoué).
- Le chargement de X par GPU2 échoue : refus d'allocation mémoire. GPU2 retourne essayer de prendre le verrou sur la donnée Y. Or, le spinlock header_lock sur la donnée Y est verrouillé par GPU1 : GPU2 part précharger de nouveau la donnée X sur GPU1 (puisque le premier chargement a échoué).
- Conséquence : livelock.

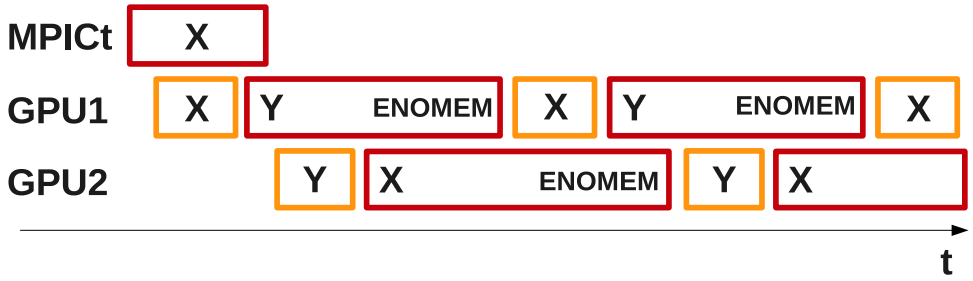


FIGURE 4.4 – Scénario de livelock

4.2.3 Un problème de livelock

Un livelock provoque l'arrêt de la progression du programme [21]. Un programme P possède un livelock si P peut atteindre un état dans lequel, après l'avoir atteint, l'exécution ne bloque jamais, ne termine jamais, mais ne progresse plus.

Dans notre cas, ce sont les phases de préchargement des données, lorsqu'un spinlock sur une donnée n'est pas disponible, qui interagissent mal entre elles et provoquent le blocage.

À chaque fois que l'exécution revient au moment de la tentative de prise du spinlock, celle-ci échoue car un autre worker GPU a pris le verrou sur cette donnée pour la précharger, du coup le worker GPU courant part faire du préchargement et prend le verrou de la donnée sur lequel l'autre worker GPU veut travailler, de sorte que lorsqu'il va revenir tenter de verrouiller le spinlock sur la donnée, il va également échouer, et repartir précharger la donnée du premier

worker GPU, et ainsi de suite. La situation se débloque lorsque, par chance, les deux workers GPU finissent par tenter de verrouiller leurs données respectives en même temps, réussissent finalement, permettant à l'exécution de se poursuivre.

Nous allons proposer deux solutions à ce problème, présenter les bons et mauvais aspects de chacune des deux, puis faire un choix et présenter les résultats obtenus.

4.2.4 Une idée : des ticket locks

La première idée que nous avons eue est la suivante : si le problème réside dans le fait que le timing de la tentative de prise du verrou est mauvais, nous allons utiliser un autre système de spinlock permettant de mémoriser les tentatives de prise du verrou, afin de laisser la priorité de reprise du verrou au thread ayant tenté de le prendre en premier. Le système que nous proposons est une adaptation du ticket lock.

Le fonctionnement du ticket lock([22]) est le suivant : à chaque tentative de prise du verrou, un ticket est déposé sur une liste interne au spinlock. Le ticket est alloué par le thread qui le dépose, et lui seul peut y accéder. Lorsque le verrou est libre et que quelqu'un tente de le verrouiller, il doit regarder la liste des tickets pour savoir s'il est prioritaire ou non. S'il ne l'est pas, il dépose alors un ticket à la fin de la liste et le verrouillage échoue.

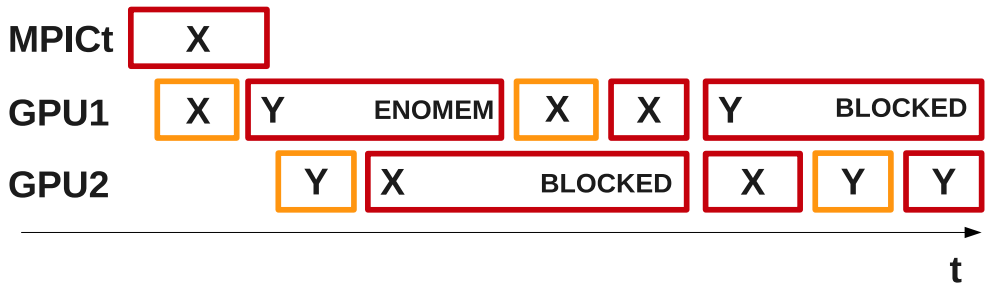


FIGURE 4.5 – Scénario souhaité avec les ticket locks

Cette solution n'est pas suffisante : nous pouvons présenter un cas dans lequel les ticket locks ne font que repousser le problème autre part, à savoir dans la liste des tickets. Reprenons l'exemple de la partie 4.2.2 :

- MPICt prend le header_lock sur la donnée X.
- GPU1 essaye d'exécuter une tâche sur la donnée X. Or, le spinlock header_lock sur la donnée X est verrouillé par MPICt : GPU1 part précharger la donnée Y sur GPU2, et dépose un ticket sur la liste de la donnée X.
- GPU2 essaye d'exécuter une tâche sur la donnée Y. Or, le spinlock header_lock sur la donnée Y est verrouillé par GPU1 : GPU2 part précharger la donnée X sur GPU1, et dépose un ticket sur la liste de la donnée Y. Or, la donnée X a un ticket précédemment déposé par GPU1 : GPU2 se bloque.
- GPU1 essaye à nouveau de prendre le spinlock sur la donnée X, mais il est encore verrouillé par MPICt : GPU1 part précharger la donnée Y sur GPU2. Or, GPU2 a déposé un ticket sur la donnée Y avant GPU1 : GPU1 se bloque.
- MPICt libère le header_lock sur la donnée X.

– Situation d'interblocage : GPU1 et GPU2 sont tous deux bloqués.

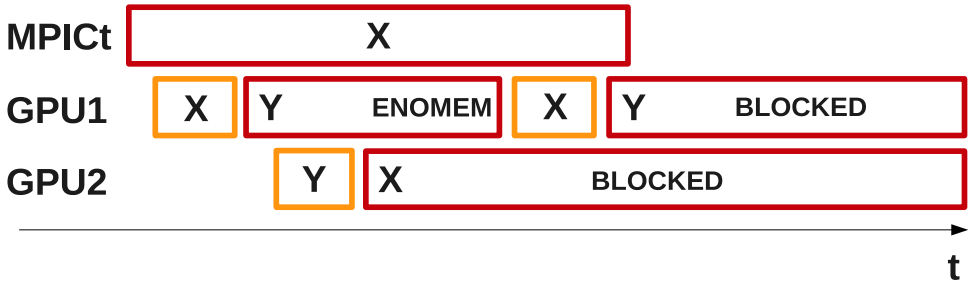


FIGURE 4.6 – Scénario réel avec les ticket locks

Il est également possible que, lorsqu'un worker GPU part faire du préchargement de données, si la tâche que le worker GPU essaye de faire est un chargement, il peut en fait s'occuper du préchargement de la requête de chargement qu'il vient de soumettre lui-même, ce qui peut causer un interblocage lors de l'utilisation de ticket locks. C'est un problème connu des ticket locks qui est résolu par l'utilisation de mutexes récursifs. Il est donc possible de le corriger, mais le premier problème reste entier.

4.2.5 Borner le nombre de trylocks avant un lock

Notre deuxième idée, et celle que nous allons retenir, est la suivante : afin de limiter l'impact de cette situation tout en permettant la progression des communications, nous avons choisi de définir une borne maximum pour le nombre d'appels au gestionnaire de progression des communications qu'un worker est autorisé à faire avant de demander un verrouillage bloquant du lock, afin d'être sûr d'obtenir le verrou sur la donnée au bout d'un certain temps, pour que l'exécution puisse progresser.

Cependant, nous nous attendons à de possibles pertes de performances, car nous enlevons la possibilité, jusqu'à un certain point, de faire progresser les communications pendant l'attente bloquante de la prise du spinlock, mais également à un gain de stabilité desdites performances dans le cadre d'exécutions sur des grappes de machines hétérogènes. Nous espérons que cette solution, à défaut de faire disparaître les livelocks, limitera drastiquement leur durée et donc leur impact sur les performances.

4.2.6 Un gain en stabilité

Voici la trace d'exécution que nous obtenons après l'application de notre solution :

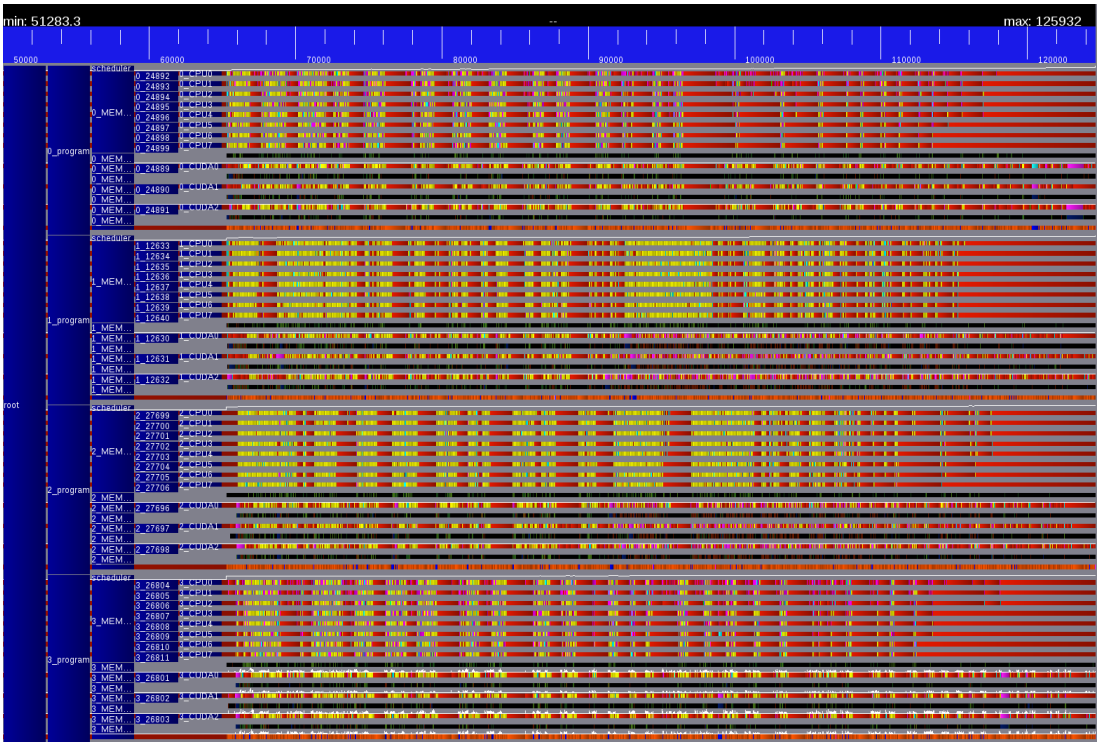
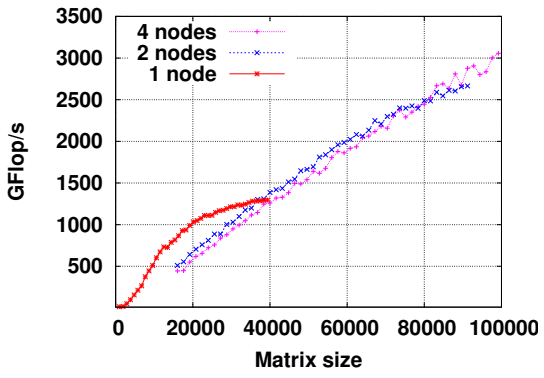
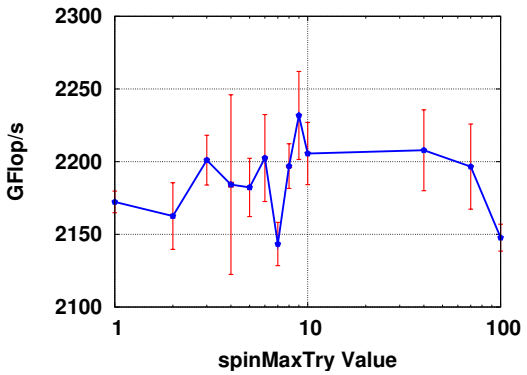


FIGURE 4.7 – Trace obtenue avec la borne de trylock à 30

Les deux figures suivantes présentent les résultats obtenus après application de notre solution :



(a) Borne de tentatives = 10, taille de bloc = 1600



(b) N=76800 , NB=960

FIGURE 4.8 – Résultats obtenus après l'ajout de la borne de trylock

La figure 4.8(a) présente les performances obtenues, qui sont désormais totalement stables. Nous observons cependant que nous sommes encore loin de la crête de performance. Ceci s'explique par le fait que, la taille du problème devenant très grande au-delà de matrices 100.000*100.000,

ce dernier ne loge plus en mémoire. Des idées d'optimisation mémoire ont été soumises à l'équipe de développement de MORSE afin de pouvoir dépasser cette limitation.

La figure 4.8(b) montre le comportement des performances avec différentes valeurs de la borne maximum de tentatives de prise de verrou. Nous pouvons constater que les performances sont instables entre 1 et 10 avec des barres d'erreur importantes, stables entre 10 et 40, mais que les performances commencent à chuter à partir de 70. Cette chute est causée par une valeur trop élevée de la borne, réintroduisant des zones de livelock d'une durée suffisamment importante pour affecter la stabilité des performances. Nous avons donc choisi, dans notre solution, de fixer cette borne à 10.

Chapitre 5

Conclusion

Notre travail s'est basé sur les trois axes suivants :

Tout d'abord, faire un point précis des performances actuelles de StarPU et StarPU-MPI, afin de mettre en lumière les problèmes de performance éventuels. Ensuite, les analyser en utilisant et/ou en améliorant les outils de visualisation d'exécution intégrés à StarPU. Pour finir, proposer des solutions aux problèmes rencontrés, étudier leur impact sur les performances, présenter les gains obtenus.

Nous allons résumer le travail effectué dans chacun de ces axes, puis évoquer les perspectives ouvertes par notre travail.

5.1 Définition d'un cadre de travail et d'expérimentation

Nous avons tout d'abord présenté le support d'exécution sur lequel nous avons travaillé, qui se nomme StarPU. Pour analyser l'exécution d'applications sur celui-ci, nous avons utilisé les événements Pajé pré-existants dans StarPU. Le format Pajé est un format générique de trace d'événements d'exécution sur machines parallèles et/ou distribuées. Pour pouvoir visualiser les traces d'exécution, nous avons utilisé ViTE, qui est un outil de visualisation de trace.

5.2 Un protocole de test basé sur MORSE

La plateforme que nous avons utilisé pour nos tests se nomme PlaFRIM. Le protocole de test que nous avons mis en place est le suivant : tout d'abord définir les variables à étudier, puis détailler le protocole que nous allons utiliser pour faire la batterie de tests et son automatisation, ainsi que le système de dossier mis en place.

Les premiers résultats que nous avons obtenus sont satisfaisants, sauf pour le test de scalabilité sur Minotaure et le test de scalabilité sur grappes de machines hétérogènes. Pour obtenir une bonne scalabilité des performances pour le test sur Minotaure, nous avons utilisé l'outil numactl et l'ordonnanceur eager.

Nous avons découvert un problème de performance de StarPU sur des grappes de machines

hétérogènes. L'un des objectifs de notre travail a été d'analyser les informations dont nous disposions afin d'essayer de le résoudre.

5.3 Un gain en fiabilité : gestion des tags

Dans les implémentations MPI, le nombre de communications simultanées sur différents tags est limité, ce qui peut causer des problèmes de scalabilité dans le cadre de leur utilisation avec StarPU.

Pour régler ce problème, nous avons proposé l'implémentation d'un système enveloppe + donnée permettant l'identification d'une donnée à recevoir grâce à son tag, contenu dans l'enveloppe qui la précède. Il nous a fallu gérer le scénario dans lequel une donnée pouvait arriver (i.e une enveloppe était reçue) alors que la réception correspondante n'avait pas encore été postée côté récepteur. Pour cela, nous avons mis en place un système de tampon temporaire stockant ces données, qui seront recopiées dans la donnée réelle quand l'application soumettra les requêtes de réception qui leurs sont associées.

Nous avons constaté lors de nos tests de validation de cette solution que le problème de performance sur des grappes de machines hétérogènes subsistait. Il était donc nécessaire de poursuivre notre recherche.

5.4 Un gain en stabilité : gestion d'un livelock

Les traces d'exécution mettaient en évidence des problèmes de transfert de données vers les GPU anormalement longs au moment d'une communication MPI. L'état des outils d'analyse et de visualisation des traces d'exécution ne nous permettait cependant pas de comprendre la nature du problème.

Dans un premier temps, nous avons suspecté un possible conflit entre des verrous. Nous avons donc ajouté des événements de trace correspondant à tout ce qui concerne des prises et relâchements de verrous. Cette analyse a permis de mettre à jour la zone de code dans laquelle boucle l'exécution. Nous avons ensuite raffiné nos événements pour analyser l'apparition de l'événement : refus d'allocation mémoire. Ces derniers ont permis la mise en évidence du problème, mais nous ne connaissions toujours pas le scénario exact de blocage. Pour le découvrir, nous avons encore affiné nos événements pour savoir quelles données étaient concernées. Nous avons alors constaté que ce sont des tentatives de prises de verrou concurrentes sur des données qui provoquaient un scénario de livelock. Notre première proposition s'est basée sur l'utilisation de ticket locks, qui est la solution classique proposée par l'état de l'art pour résoudre les livelocks. Cependant, cette solution ne faisait que reporter le problème dans la liste des tickets associés aux verrous, et pouvait même provoquer un interblocage. La solution que nous avons retenue est de borner le nombre de tentatives de verrouillage avant de prendre le verrou de manière bloquante, afin de limiter au maximum la durée des livelocks tout en permettant une progression suffisante des communications.

Cette solution nous a permis d'obtenir des performances stables sur des grappes de machines hétérogènes, et donc de résoudre le problème d'instabilité des performances.

5.5 De nombreuses perspectives

5.5.1 Prioriser les requêtes

Les listes actuellement utilisées dans StarPU-MPI pour stocker les communications à soumettre, ou à traiter, notamment dans le cadre des communications détachées, sont de simples listes FIFO, qui sont toujours parcourues dans l'ordre.

Or, il arrive qu'une réception sur une donnée nécessaire à l'exécution d'une tâche bloquant le reste de l'exécution (typiquement un POTRF de la factorisation de Cholesky) soit postée en dernier, et donc traitée en dernier par le thread de progression à cause de ce système de liste, ce qui peut ralentir le programme.

Une idée pourrait être d'introduire des listes de priorité ou de la réorganisation de listes, afin de prioriser le traitement des réceptions dont la terminaison rapide est nécessaire pour permettre la progression de l'exécution.

5.5.2 Une réception avec allocation au dernier moment

Les données associées à une communication sont actuellement allouées à la soumission de la requête par l'application via le starpu_data_acquire_cb, et non pas au moment où la donnée arrive réellement, ce qui peut poser des problèmes de limitation mémoire sur des machines à mémoire réduite, ou pour des applications appliquant une grosse pression sur la mémoire des machines.

Il pourrait être intéressant d'implémenter une fonction de réception dédiée à ce besoin, permettant de n'appeler le starpu_data_acquire_cb qu'au moment où le thread de progression sait que la donnée correspondant à la requête de réception est arrivée.

5.5.3 Déroulement du graphe de tâches : « pruning »

Actuellement, chaque instance de StarPU-MPI déroule l'intégralité du graphe de tâches, afin de découvrir les tâches qu'il doit effectuer, ainsi que les émissions / réceptions qu'il sera nécessaire de poster durant l'exécution. Cependant, ce déroulage est un facteur limitant pour la scalabilité, à partir du moment où le graphe de tâches devient très grand.

Une idée serait de faire du "pruning", i.e de découper le graphe de tâches, de sorte que chaque instance de StarPU-MPI n'ait connaissance que du sous-graphe de tâches qui lui est nécessaire, ce qui permettrait de gagner en scalabilité.

Ce problème a été étudié dans le support d'exécution PaRSEC, qui a proposé une solution à base de graphe de tâches algébrique. De futurs travaux peuvent apporter cette fonctionnalité dans StarPU.

5.5.4 Agréger les transferts de données

Nous avons observé sur les dernières traces d'exécution la présence d'un phénomène affectant la durée des communications lorsqu'un nœud soumet un grand nombre de requêtes d'émission dans un délai très court : plus la communication a été soumise tard, plus le temps de complétion de la communication est long.

Une idée pourrait être, lorsque c'est possible, qu'un nœud puisse détecter qu'il a soumis de nombreuses requêtes d'émission vers un unique nœud N, afin d'agréger les multiples transferts de données en un seul.

Ce ne sont que quelques-unes des propositions qu'il est possible de faire pour améliorer la scalabilité de StarPU sur des grappes de machines hétérogènes, qui reste un problème ouvert et passionnant.

Bibliographie

[1] Francisco D Igual, Ernie Chan, Enrique S Quintana-Ortí, Gregorio Quintana-Ortí, Robert A Van De Geijn, and Field G Van Zee. The flame approach : From dense linear algebra algorithms to high-performance multi-accelerator implementations. *Journal of Parallel and Distributed Computing*, 72(9) :1134–1143, 2012.

[2] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin : exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 45–55. IEEE, 2009.

[3] Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, Bruno Raffin, et al. Xkaapi : A runtime system for data-flow task programming on heterogeneous architectures. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2013.

[4] Jie Chen, W Watson, and Weizhen Mao. Gmh : A message passing toolkit for gpu clusters. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 35–42. IEEE, 2010.

[5] Jinpil Lee and Mitsuhisa Sato. Implementation and performance evaluation of xcalablemp : A parallel programming language for distributed memory systems. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 413–420. IEEE, 2010.

[6] Jinpil Lee, Minh Tuan Tran, Tetsuya Odajima, Taisuke Boku, and Mitsuhisa Sato. An extension of xcalablemp pgas lanaguage for multi-node gpu clusters. In *Euro-Par 2011 : Parallel Processing Workshops*, pages 429–439. Springer, 2012.

[7] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. Dague : A generic distributed dag engine for high performance computing. *Parallel Computing*, 38(1) :37–51, 2012.

[8] David M Kunzman and Laxmikant V Kalé. Programming heterogeneous clusters with accelerators using object-based programming. *Scientific Programming*, 19(1) :47–62, 2011.

[9] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on*, pages 394–401. IEEE, 2009.

[10] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M Badia, Xavier Martorell, Eduard Ayguade, and Jesus Labarta. Productive programming of gpu clusters with ompss. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 557–568. IEEE, 2012.

[11] Cédric Augonnet. *Scheduling Tasks over Multicore machines enhanced with acelerators : a Runtime System's Perspective.* PhD thesis, Université Bordeaux 1, 2011.

[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 23(2) :187–198, 2011.

[13] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. Starpu-mpi : task programming over clusters of machines enhanced with accelerators. In *Recent Advances in the Message Passing Interface*, pages 298–299. Springer, 2012.

[14] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, Stanimire Tomov, et al. Faster, cheaper, better–a hybridization methodology to develop linear algebra software for gpus. *GPU Computing Gems*, 2, 2010.

[15] Emmanuel Agullo, George Bosilca, Berenger Bramas, Cedric Castagnede, Olivier Coulaud, Eric Darve, Jack Dongarra, Mathieu Faverge, Nathalie Furmento, Luc Giraud, et al. Matrices over runtime systems at exascale. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion :*, pages 1330–1331. IEEE, 2012.

[16] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures : The plasma and magma projects. In *Journal of Physics : Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[17] Asim YarKhan, Jakub Kurzak, and Jack Dongarra. Quark users' guide : Queueing and runtime for kernels. *University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02*, 2011.

[18] B de Oliveira Stein, J Chassin de Kergommeaux, and G Mounié. Pajé trace file format. Technical report, Tech. rep.(March 2003), 2010.

[19] Kevin Coulomb, Augustin Degomme, Mathieu Faverge, and François Trahay. An open-source tool-chain for performance analysis. In *Tools for High Performance Computing 2011*, pages 37–48. Springer, 2012.

[20] MPI Forum. Mpi 2.2 standard - message passing interface forum, 2009. [Online ; accessed 3-June-2013].

[21] Kuo-Chung Tai. Definitions and detection of deadlock, livelock, and starvation in concurrent programs. In *Parallel Processing, 1994. ICPP 1994 Volume 2. International Conference on*, volume 2, pages 69–72. IEEE, 1994.

[22] John M Mellor-Crummey and Michael L Scott. Synchronization without contention. *ACM SIGPLAN Notices*, 26(4) :269–278, 1991.